LA-UR-18-29608
Approved for public release; distribution is unlimited.

Title:	Learning Memento archive routing with Character-based Artificial Neural Networks
Author(s):	Powell, James Estes Jr.
Intended for:	Report Web
Issued:	2018-10-10

Learning Memento archive routing with Character-based Artificial Neural Networks

James Powell

November, 2017

Abstract

This white paper describes a series of tests that were performed to determine if a neural network could learn patterns from a service that maintains a cache of routing decisions for the discovery of version information for discrete web-at-large URLs. Labeled training data was derived from a log file that records the by-archive availability of Memento availability for a given URL. Training data sets were generated from this log file on a by-archive basis (thus making it a binary classification problem). Each training data set consisted of equal numbers of hit and miss URLs, selected at random. The URLs were converted to a normalized numeric representation where each integer in a URL training vector represents a character in that URL. The corresponding label indicates whether or not Mementos were available for that URL in the selected archive. The training data matrix and the label vector became input to a neural network. A number of neural network architectures and network hyperparameters were explored, however the log entries themselves were used as-is, without any feature engineering, beyond the aforementioned normalization.

The paper is divided into five major sections. The first section describes the general conceptual model underlying artificial neural networks. The next section steps through the execution of a very simple neural network, implemented in python. Following that, the paper introduces two common network architectures, multi-layer perceptrons and convolutional neural networks. Next is an expansion on the problem statement, followed by a characterization of the test data set and details about how test data is transformed into labeled training data. This section introduces the concept of hyperparameters which are influenced by various aspects of the training data. The results section presents details about the final versions of the network architectures which were tested, and the results of those tests. The paper concludes with a preliminary evaluation of the performance of a learned model, and suggests future work. A processing pipeline that resulted from this work appears in table 8. Additional results are included in the appendices.

Overview of Neural Networks

Artificial Neural Networks (ANNs), or neural networks, are a computational form of biomimicry. In living organisms that possess a nervous system, neurons are the information signaling building blocks which handle sensory input. The primary role of neurons is to receive and transmit signals to one another under certain conditions. A neuron sums the inputs it receives, and forwards the input if a certain threshold value is reached. When an organism feels pain, nerves local to the source of pain will relay this signal onward. If the sensation is sufficiently intense, this continues until the signal reaches the brain, where it may cause a reaction and/or the formation or reinforcement of a memory. Artificial neural networks approximate this process.

Instead of sensory input, ANNs are exposed to vectors of data. An ANN can learn from labeled or unlabeled data. An example analogous to how ANNs learn from labeled data is language learning. For this task, one might use flash cards, which combine an image of an object with the corresponding text representing the word for that image. When viewing a card, the images are turned into impulses by the retina, passed along the optic nerve, and ultimately to the brain. Over time the association between word and its visual representation becomes stronger. Artificial neural networks learn in a similar fashion. The network is exposed to training data and corresponding labels repeatedly. As this happens, a set of weights, initialized to random values, are progressively tuned so that they model the relationships between training data and labels. The resulting model can be then used to identify similar data and make label predictions about it. Prediction from a model is similar to how a language learner might guess the meaning of a new word they had not previously seen, based on the fact that it shares a root with a word they have previously learned.

Neural networks are arranged as layers of neurons. Each neuron receives some input data and performs some mathematical operations on that input. Every neuron in a given layer is computationally identical to all others in that layer. Neurons within a layer are not connected to one another, but they are connected to neurons in other layers. Those other layers contain neurons that may use different operations to process the input they receive. Often all neurons between two layers are completely connected but there are network architectures where this is the exception rather than the rule. ANNs are data-agnostic. The network has no prior information about the data to which they will be exposed. When a neural network has many layers, it is classified as a deep network. The next section is a look under the hood of a very simple neural network implementation.

Simple neural network walk through

The best way to understand neural networks is to examine the implementation and output of a simple example. The blog 'Trask' by iamtrask published a bare-bones neural network (below) that performs binary classification with labeled data. It uses just nine lines of python code. The only external library it relies on is numpy, a math library for python. Neural networks are usually built using specialized libraries, such as TensorFlow and Keras for example, but these libraries abstract away some of what is actually happening. Trasks stripped down example explicitly exposes every step. The input data is a matrix of numbers that represent the training data (X), and a corresponding vector of class labels (y). A single layer of neurons is exposed to this data. To borrow again from the biological analog, this layer is where signals get aggregated and compared to a threshold before they are adjusted and relayed onward (in this case, back through the network). Although it is a toy example, it does illustrate many of the basic concepts of neural networks.

iamtrask numpy neural network example, in its entirety, with comments added for clarity:

```
X = np.array([ [0,0,0,1],[0,0,1,1],
               [0,1,1,1],[1,0,0,0] ])          # 1. Training data, matrix X
y = np.array([[0,0,1,1]]).T                    # 2. Labels for training data, vector y
syn0 = 2*np.random.random((4,1)) - 1           # 3. Initialize random weights, syn0
for j in xrange(10000):                        # 4. An iteration: feedforward,
                                               #    apply weights to each training
                                               #    vector in training matrix X
  l1 = 1/(1+np.exp(-(np.dot(X,syn0))))         # 5. Use sigmoid function to predict a label
                                               #    for each vector in X where x in e^-x
                                               #    is dot product of each row in X
                                               #    combined with the weights in syn0
  l1_error = y - l1                            # 6. Calculate error: labels - predictions
  l1_delta = l1_error * (l1*(1-l1))            # 7. Use partial derivative of l1 multiplied
                                               #    by l1_delta to determine if predicted label
                                               #    is moving toward or away from y labels
  syn0 += X.T.dot(l1_delta)                    # 8. Now adjust the weights
  print l1                                     # 9. print predictions for this iteration
                                               #    Repeat from line 4 onward until j=10000
```

The input training data:

$$X = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

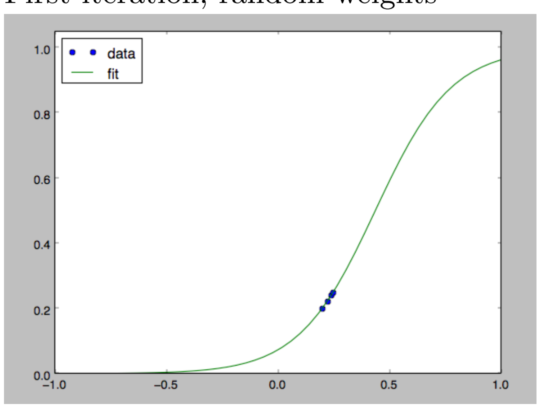
The input classes vector (row in X "is a class of" entry in y):

$$y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Table 1 shows the output from each step performed by the network. In the left column, the first entry is a set of random weights. These were generated by the network when it initialized itself for the first time. The weight vector is combined with each of the training vectors by computing their dot product. This value represents the combination of the direction and magnitude of a training vector with the weights vector. Calculating the dot product of these values is the first step performed by each neuron. The dot product is an input value to a function associated with the node. It is this function that generates a prediction based on these inputs. This function is referred an activation function. The activation functions associated with neural network nodes and the edges between layers of nodes are the defining characteristics of neural networks. This example uses the sigmoid (also known as the logistic sigmoid) function as its activation function. A plot of the sigmoid function has a characteristic sideways S shape (figure 1). For any f(x), the result will fall somewhere on the sigmoid curve. The sigmoid function tends to output values that are close to 0 or 1. This is why it works well as an activation function for binary classification. The results of these operations produce a set of predictions for a label, given a row in X. Once each training vector is combined with the weights vector, these values are used as value x for the sigmoid function. This is an example of how a neural network uses an activation function to learn nonlinear patterns in labeled training data.

$$f(x) = \frac{1}{1 + e^{-x}}$$

First iteration, random weights



Weights after 1000 iterations

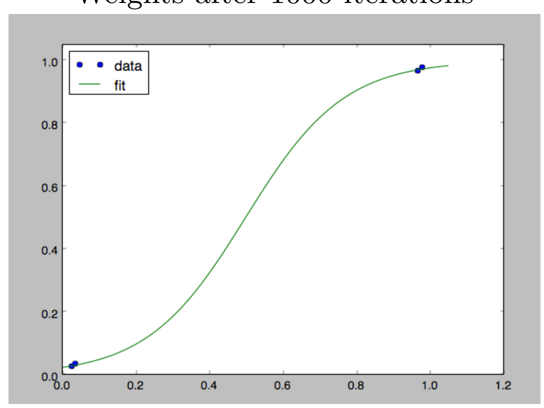


Table 1: Sigmoid plot of weights

The right column of table 1 illustrates the next steps. These steps evaluate the predictions for their ability to predict a class in vector y. The difference between the partial derivative of a predicted value and the difference between a label and the predicted value (error) is called the error delta. The error delta is a vector of values used to update the weights. This process of combining the training data vectors with the weights vector, generating predictions, evaluating the predictions in relation to the assigned labels, and then readjusting the weights is how the network gradually learns an association between a vector of input data and its assigned label. Examples of predictions at various iterations are provided in table 2.

initial random weights	$\begin{bmatrix} -0.16595599 \\ 0.44064899 \\ -0.99977125 \\ -0.39533485 \end{bmatrix}$	error	$\begin{bmatrix} -0.40243371 \\ -0.19859385 \\ 0.72201066 \\ 0.54139404 \end{bmatrix}$
dot product $X \cdot weights$	$\begin{bmatrix} -0.39333485 \\ -1.39510611 \\ -9.95445712 \\ -0.16595599 \end{bmatrix}$	error delta	$\begin{bmatrix} -0.09677759 \\ -0.03160707 \\ 0.14491567 \\ 0.13442085 \end{bmatrix}$
sigmoid (guess)	$\begin{bmatrix} 0.24048082 \\ 0.15915433 \\ 0.20071127 \\ 0.24828653 \end{bmatrix}$	updated weights	$\begin{bmatrix} -0.03153514 \\ 0.58556466 \\ -0.88646265 \\ -0.37880384 \end{bmatrix}$

Table 2: Results of one iteration through the network

0th iteration guess	$\begin{bmatrix} 0.40243371 \\ 0.19859385 \\ 0.27798934 \\ 0.45860596 \end{bmatrix}$	100th iteration guess	$\begin{bmatrix} 0.09428112 \\ 0.10927185 \\ 0.88435938 \\ 0.91770372 \end{bmatrix}$
results of 10,000 iterations	$\begin{bmatrix} 0.00727093 \\ 0.01015224 \\ 0.98981649 \\ 0.99282528 \end{bmatrix}$	target classes to guess	$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$

Table 3: Progress at various iterations

Character-based Neural Networks

Neural networks are used for a variety of text processing tasks, including sentiment analysis, classification, summarization, topic extraction, and even artificial text generation. There are two ways to present textual data to neural networks, as words or as characters. In either case, the data is transformed into a matrix of numbers before it is presented to the network. Of the two approaches, utilizing characters as features requires less data preparation. Basically a dictionary of discrete characters is created from the text, and string values are transformed into a vector of indices into this dictionary. The normalized data vectors, along with the training labels, becomes the input to a neural network.

A multi-layer perceptron (MLP) network uses multiple layers to learn relationships in a training data set. Like the python example above, it initializes itself with random weights, and accepts as input a set of vectors that represent training data, together with their corresponding labels. It includes additional layers, referred to as hidden layers, which can improve the networks ability to learn. Hidden layers can have different numbers of nodes and use different techniques for processing the data than the first layer. For binary classification tasks, the final layer will only contain one node. That node will utilize an activation function that reduces the final output to value of 1 or 0. Sigmoid is usually used as the activation function for this layer. Much effort is devoted to fine tuning aspects of the hidden layers. Many factors affect the design of these layers, and so sometimes they are determined experimentally.

The term hyperparameters refers to various changeable aspects of a neural network architecture. These include how much training data is presented to a given node in the first layer, how many iterations the network revisits the data, and other characteristics such as the way in which loss (error) is evaluated, the number of nodes per layer, and the type of activation functions used by discrete layers. For neural networks that treat individual characters as features, some important hyperparameter values to consider are length of document, number of distinct characters in training data, and batch size.

Convolutional neural networks (CNNs) are more complex than MLP networks. They have been most commonly applied to image modeling and object recognition tasks. There are a couple of reasons why they are especially well suited to this task. CNNs can process multi-dimensional data effectively. Images typically have additional dimensions such as location, color, and brightness depending on the source files. CNNs break up the input data into smaller regions, using a fixed size window that it can move across the data. This is called a kernel filter. The process of moving the kernel filter over input data is called convolution. Convolution over image data is especially good at detecting distinctive features, such as edges and outlines. CNNs also perform better with many layers because initial layers can detect small features and subsequent layers (pooling layers and/or layers with different kernel sizes and number of nodes) can aggregate these features. CNN models can learn to classify images, they can be used to recognize images of characters from a page scan, objects in a photograph, or faces captured by a security camera. It turns out that convolutional neural networks are also good at learning over character vector representations of textual data.

In the spring of 2017, the Textrodeo team applied a number of machine learning algorithms and neural network architectures to the problem of identifying email spam. This is a binary classification problem so labeled data consists of a training set of the text of spam and non-spam emails, together with a label indicating their class. Character CNNs were among the top performers of the algorithms tested (see figure 2 and 3). They perform well using the raw text as training data. This was the inspiration for applying MLPs and CNNs to Memento aggregator log file data, since it could be framed as a binary classification problem. The remainder of this white paper describes how these neural network architectures were constructed and tested with aggregator cache data, and the results produced by these networks.

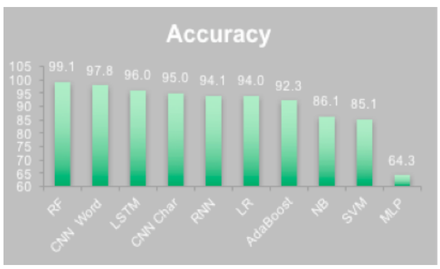


Figure 1: Comparison of accuracy achieved by ML and NNs on spam classification problem

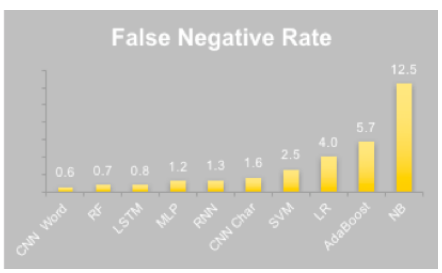


Figure 2: Comparison of false positive rates of classical ML algorithms and NNs for spam classification

Time Travel for the Web

Web archives store snapshots of Web content collected at a particular moment in time. Sometimes archiving happens automatically, while other times it is triggered by an event or a user. The archive stores each object together with the URL used to reference it, and a timestamp. An item in a Web archive can be retrieved by constructing a URL that includes this time component. The archive will respond with the object that was archived at that time, or the last version of the object that was archived prior to the specified time. Individual web archives are searchable, and if the content requested is a Web page, the URLs within that Web page are rewritten so that a complete approximation of that Web page is returned for the timestamp.

There is no one archive that maintains versions of all content on the Web. The oldest Web archive service is the Internet Archive at `https://web.archive.org/web/`, so it tends to be the most reliable and comprehensive source for snapshots of Web content. Different organizations have launched their own Web archives over time, with different coverage goals. Newer archives obviously do not cover time periods before their existence, but since they may have different coverage of Web-at-large content, they may over time become better sources for requests for more recent content. In short, Web archives have grown in numbers and coverage organically, so today what can be found of Web content from the past is a patchwork of digital objects. It is a situation that is improving constantly. But meanwhile providing an optimal user experience for accessing Web archive content has required the development of a new protocol (Memento) and services that use that protocol.

The Memento protocol is published as an RFC with the IETF. The RFC is entitled "HTTP Framework for Time-Based Access to Resource States – Memento." The standard was built on existing capabilities of the HTTP Web protocol. It allows a resource to provide information about other versions of itself that might exist (Mementos), and the source for those versions (a Memento Timegate. The temporal Web search engine Time Travel for the Web searches collections of Mementos for Web resources. Since there are multiple archives that may contains Mementos for a resource, this is a federated search.

Heres a portion of the response header of a request to archive.is for past versions of a website called www.dannydorling.org:

```
<http://www.dannydorling.org/>; rel="original", \\
 <http://archive.is/timegate/http://www.dannydorling.org/>; rel="timegate",
 <http://archive.is/timemap/http://www.dannydorling.org/>; rel="timemap";
 type="application/link-format";
 from="Sun, 14 Apr 2013 14:56:09 GMT"; until="Sun, 14 Apr 2013 14:56:09 GMT",
<http://archive.is/20130414145609/http://www.dannydorling.org/>;
rel="first last memento";
datetime="Sun, 14 Apr 2013 14:56:09 GMT"
Memento-Datetime: Sun, 14 Apr 2013 14:56:09 GMT
```

This example shows that archive.is does indeed have previous versions of the specified website and it provides some additional information about what it has.

As with many archived URLs, other archives also have previous versions of this website. Thus finding all the possible sources for versions of this web site is a federated search task. The Memento aggregator sits between Memento-based services and the archives that support it. It maintains information about which archives have version information for a given URL, but it tries to strike a balance between optimizing federated archive queries but not maintaining a complete list of each archives holdings, which is not practical. The aggregator maintains a cache of recently queried items to improve response time. The aggregator log is a good indicator of which archives are able to respond to Memento requests for which URLs. But a cache obviously favors more frequently and more recently requested items. Finding other strategies to allow the aggregator to decide which archives to target can improve the performance of the Aggregator. Some strategies that have been employed including heuristics that leverage domain names, and the application of various machine learning algorithms. Following in the footsteps of these optimization efforts, this white paper explores the possibility of using various types of ANNs together with a sample of aggregator log file entries to learn models that could optimize aggregator routing.

Using character-based neural networks to learn to route Memento aggregator requests

The test dataset consisted of data from Memento aggregator logs. Subsets of this data were used as labeled training data to test the ability of various neural network architectures to perform binary classification tasks. There were 193,288 distinct URLs each of which is accompanied by a comma separated list of archive ids. The occurrence of an archive id with a URL indicates that the archive had information about the URL (a hit). Although a total of 39 distinct archives were represented in this dataset, only twelve occurred frequently enough to be included in the tests. Each archive in this set occurred at 1000 times in the log file. Two other archives which fell below this threshold were used for testing and development, because they could be processed more quickly. In order to construct a test set for binary classification, two lists were created one with hits, and one with misses. Each of these lists were then randomly shuffled using np.random.shuffle. Following this, an equal number of hit and miss URLs were inserted into a vector of training examples (X_train). A corresponding labels vector was generated which indicated whether each training example was a hit (true) or a miss (false) (y_train).

Although no feature engineering was required, it was necessary to characterize the data set in order to assign optimal values to some hyperparameters for the networks. Characters represent the features in the data. The following characters were identified in the log file:

```
0123456789
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

`./_-?=:&,˜+()\%$;#*!\'@`

One hyperparameter, maximum document length, established a fixed length for the training examples. URL length ranged from 7 to 908 characters. The average URL length was 46 characters. The longest URL in the log file was 908 characters. Past lengths of 200, URL length dropped quickly. As figure 5 illustrates, a cut off of 200 might also be reasonable. To ensure the network sees as much data as possible per URL, and since there were some extreme outliers in terms of length, a maximum length value of 250 was selected. Setting this value too high results in most of the input URLs being 0-padded, increases computation time while capturing no additional information. Setting it too low would almost certainly cause the network to view some URLs as identical.

Here is an example of a URL and how it appears in the training matrix:

`www.openbriefing.org/publications/report-and-articles/united-states-caught-off-guard-ira`

```
[32 32 32 63 24 25 14 23 11 27 18 14 15 18 23 16 63 24 27 16 64 25 30 11 21
 18 12 10 29 18 24 23 28 64 27 14 25 24 27 29 66 10 23 13 66 10 27 29 18 12
 21 14 28 64 30 23 18 29 14 13 66 28 29 10 29 14 28 66 12 10 30 16 17 29 66
 24 15 15 66 16 30 10 27 13 66 18 27 10 23 64  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
```

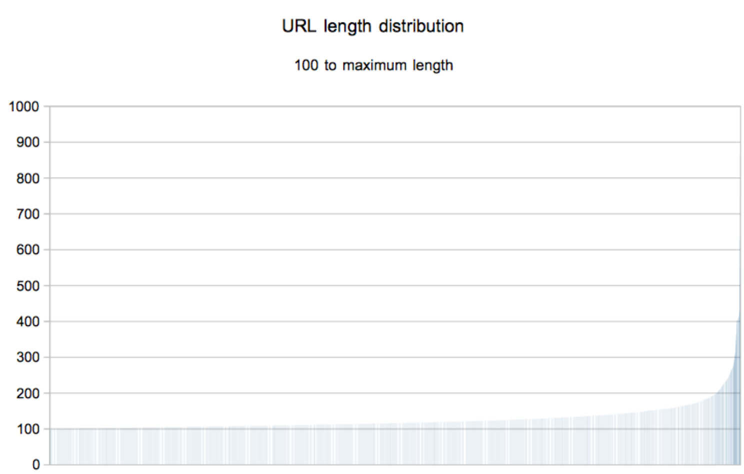


Figure 3: URL lengths

The input data for neural networks is a matrix of training data, typically containing one or two dimensional representations of each training data entry, and a corresponding vector of labels. For this project, URL strings were mapped from UTF-8 encoded characters to sequential integer value vectors (as illustrated above). These vectors were 0 padded if needed, to ensure that all were the same length. A corresponding vector of labels, containing either a 1 (hit), or 0 (miss) for each training entry, was included with the training data. One hot encoding was considered as an alternate encoding, but a literature review indicated that it was more applicable to multi-label classification problems and for encoding training data where features were tokenized words rather than discrete characters.

All neural network models for this project were implemented with the Keras python library. Keras runs on top of the popular Google TensorFlow library. It provides a layer of abstraction that corresponds more closely to a given network architecture. Here is an example of keras code that sets up a three layer CNN (the embedding is the training data input layer and so is not counted):

```
model = Sequential()
model.add(Embedding(num_chars, X_train.shape[0], input_length=X_train.shape[1]))
model.add(Conv1D(256, kernel_size=3, input_shape=(MAX_DOCUMENT_LENGTH, num_chars)))
model.add(GlobalAveragePooling1D())
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
   optimizer='rmsprop',
   metrics=['accuracy'])
```

Five different CNN architectures were tested (see table 5), with some variations in hyperparameters (mainly batch size, and number of nodes per layer). Three variations on a Char-MLP model were evaluated, and the one which performed the best was then used in comparisons with the Char-CNNs.

The batch size parameter determines how many samples each node in the neural network will see from the input. In the python example above, a technique called full batch processing was used. This approach combines all training examples with all weights. Returning to the flash card analogy, a batch size represents a subset of the cards rather than all the cards in the deck. Different batch sizes are dictated by input data and network architecture. For some applications, smaller batch sizes tend to work better because the network has a finer grained view of the data set. But this comes at the cost of performance. So identifying an optimal batch size requires some experimentation. An exceptionally large batch size, as when all nodes see most or all of the data tend to lead to poor results. The window each node has into a batch of samples is also small. Larger values for these parameters can lead to overfitting. Overfitting occurs when the model too closely matches the input data. The end result is the network cannot make good predictions when presented with previously unseen data. Another parameter that requires some trial and error is the number of epochs, that is, the number of times the network will be presented with the input data.

Given that some parameters choices could only be arrived at via trial and error, a wrapper for various CNN architectures was implemented which could

Hyperparameter	Values Tested	Selected
activation function	relu,tanh,sigmoid,softplus,sigmoid,linear,softmax,elu	relu, sigmoid
batch size	small ranges (1 -32)	32
	higher fixed and variable value (1000, numSamples/2)	1000 (Char-CNN), numSamples/2 (Char-MLP)
optimizer	adam, adagrad, sgd, rmsprop	rmsprop
loss	binary_crossentropy, mean_square_error, cosine_proximity	binary_crossentropy
kernel filter	for smaller layers: 2,3,4,5	2 (Char-CNN A), 3 (Char-CNN B, E)
	For stacked layers 9,12,15	9 (Char-CNN C, D)
epochs	10, 20, 50, 100, 300	20

Table 4: List of hyperparameters and values tested.

explore the hyperparameter space. This allowed for testing of many different combinations of parameters. Epoch values between 5 and 35, at five epoch increments were tested. Batch sizes from 32 to 144 in increments of 16 were tested for each epoch value. Kernel sizes of 3 and 4 were tested with each epoch and batch size, since prior experimentation had determined values greater than 4 yielded poorer results. However, for deeper networks, a larger kernel value in initial layers, followed by a smaller value in later layers also yielded good results. Other parameters were considered as possible candidates for experimentation, such as activation functions, optimizers, and loss functions. However early results clearly showed that relu (Rectified Linear Unit, or Rectifier) and rmsprop, respectively, which were often used in published CNN architectures, yielded the best results. Relu is the most popular activation function for deep neural networks, because it is considered to be a good representation of the biological process it approximates. Rmsprop is one of many optimization algorithms used in neural networks to reduce the error with respect to weights. It has the advantage of working well with smaller batches of training data. It is nonlinear, which is crucial as the neural network is attempting to learn linear non-separable data, but presumably data in which some sort of pattern(s) do occur. Links to further information about activation functions and optimizers are included at the end of this paper.

hit	`www.oreillynet.com/xml/blog/2006/06/why_the_world_is_ready_for_the.html`	4,52
miss	`www.npr.org/2012/03/26/149404846/the-birth-of-silicon-valley`	4,10,12

Table 5: Examples of URLs that would be labeled as true and falses when archive id 52 is specified

A problem not apparent in ROC curves is overfitting on training data. Overfitting occurs when the network does too good of modeling the training data. A plot of the training history will often reveal overfitting as a steady or dramatic divergence between training accuracy and validation data accuracy. It will also manifest as poorer performance of the model on random input data as compared to reported accuracy at training time. It turns out that larger batch sizes can help mitigate overfitting with this data set:

The hyperparameter search generated a large amount of data. The maximum number of permutations of parameters results in 84 sets of results per archive. Despite efforts to limit the sample size, execution times on a general purpose computing platform were prohibitive, with execution taking many hours

Archive id	Archive Name	Number of occurrences
4	ia	175149
7	pt	25292
8	uknationalarchives	3186
10	archiveit	34189
12	archive.is	84125
15	loc	24762
16	swa	4966
19	proni	1199
51	ukparliament	5579
52	ba	41349
71	yorku	509
82	nli	6434
86	perma	1353
110	bsb	458

Table 6: archives together with number of occurrences in log file

and sometimes days (on a quad-processor Mac Pro with 8Gb RAM and 1Tb SSD storage). ISR-3 transferred a GPU processor (Nvidia Quadro K2200) to the library for neural network projects. When this system was used, processing times were reduced by at least one order of magnitude. However, in contrast to the desktop system, which had access to most of the system RAM, the GPU unit can only utilize its onboard RAM which was 4Gb. This resulted in the execution ending prematurely with a "resources exhausted" message. So additional trial and error was required to identify the maximum sample size and network layer size that the system could handle. Because of this, the GPU system proved not to be as useful for searching the hyperparameter space, or for running tests with very large batch or sample sizes, or with deeper networks. This was especially problematic with convolutional neural network architectures because they consume more resources than simpler architectures. There are higher performance GPU services available elsewhere at LANL and from cloud services such as Amazon EC2 Elastic GPUs.

Neural networks tend to require long periods of time to run to completion. This is why it is so common to use GPU systems to execute neural networks with large labeled and unlabeled training datasets. Because of the limitations of the GPU system provided by ISR-3, and also due to variations in the maximum number of samples per archive in the provided aggregator cache log file, many of the results reported in the following section are based on relatively small numbers of samples. Most tests were run with less than 10,000 training samples per archive. There are only a few examples that illustrate the effects of larger sample sizes and/or large batch sizes.

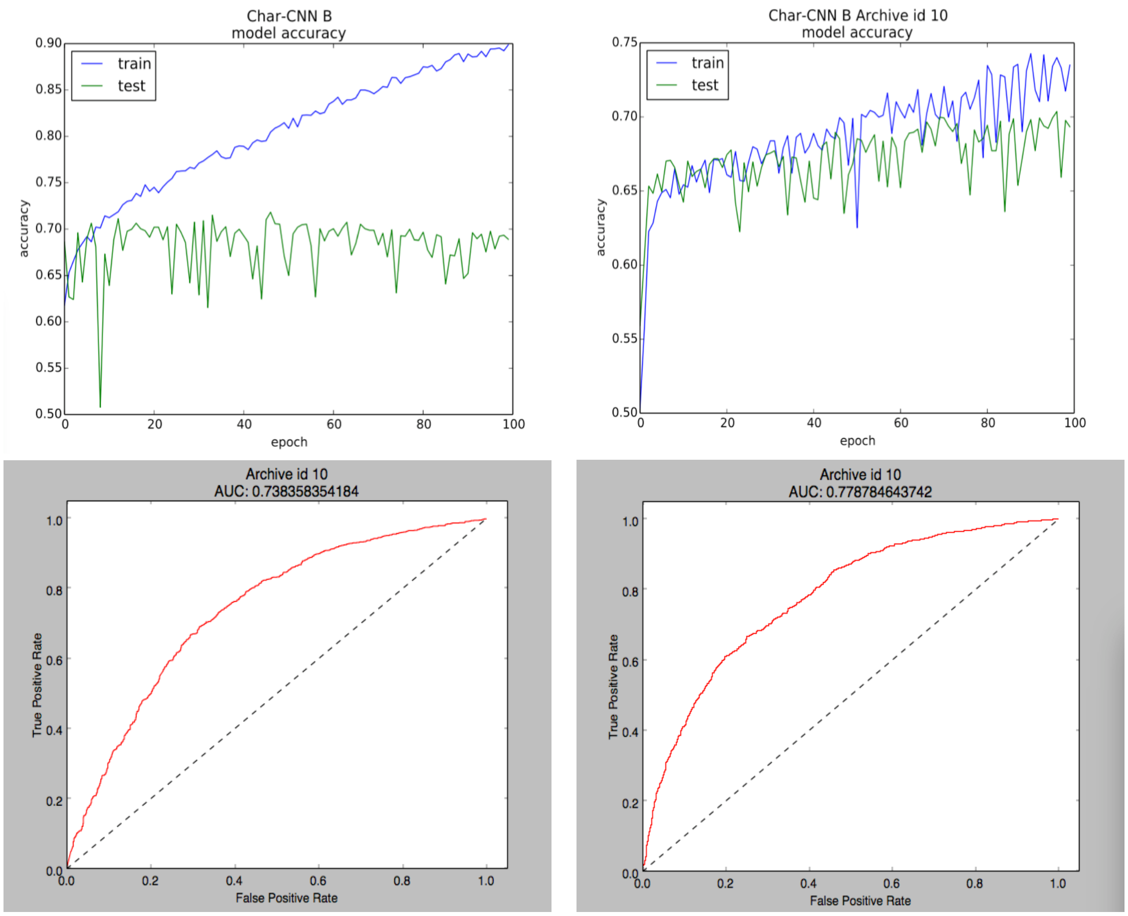
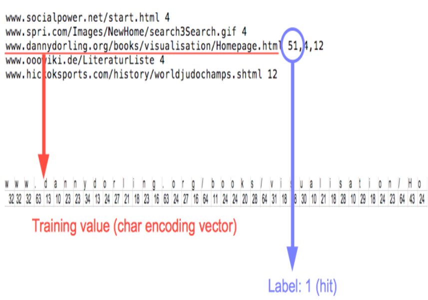
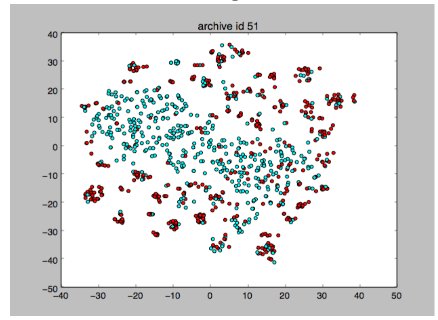


Table 7: Char-CNN with batch size of 32, compared to batch size of 1000.

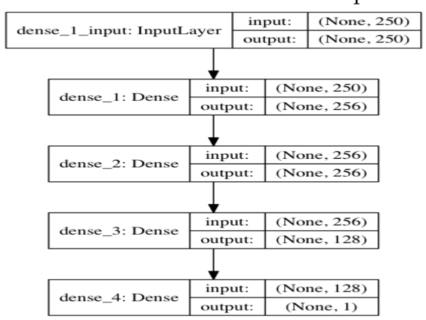
Convert log data to binary labeled training data set



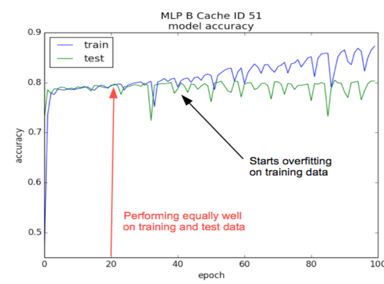
Visualize labeled training data with t-SNE



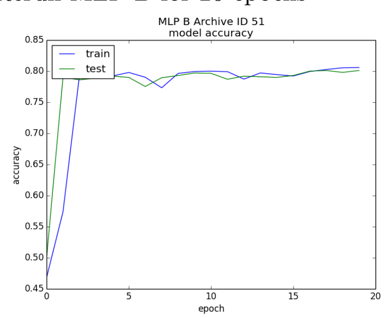
Train MLP-B network for 100 epochs



Review training history

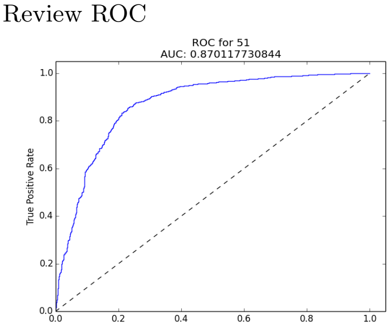


Rerun MLP-B for 20 epochs



Compare loss trend plot with training plot

Review ROC



Tests of the predictive accuracy of this model

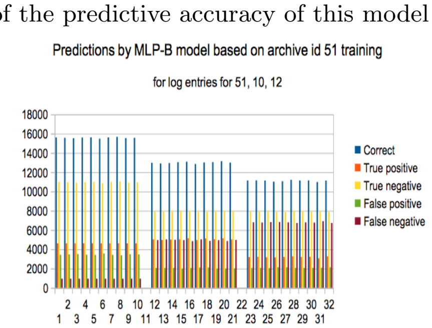


Table 8: Visual summary of NN processing pipeline for Memento aggregator log data

Results

Results include evaluation of various models against archive-specific training data, tests of various architectures and varying sample sizes. Table 7 (above) illustrates another result: a processing pipeline for applying neural networks to log files that contain URLs, when the goal of classification is to predict binary labels for that data. Other results show how optimal architectures were identified for more exhaustive testing. For example, among MLP models tested, MLP-B produced good models and with larger batch sizes (up to 50% of sample size), and little evidence for overfitting in the training history. So of the Char-MLP architectures, only Char-MLP B was used for the remainder of the tests. However, since not only the number of layers but also the type of layers as well varied among the Char-CNN architectures, all six were tested. A set of ROC curves illustrate the best results (largest area under curve) produced by a Char-CNN architecture compared with the best results from the Char-MLP for samples from the same archive. As expected, Char-CNNs yielded better results but executed more slowly, particularly if the network had many layers. Char-MLP execution time was much faster, but the results were not as good. Most hyperparameters were fixed for all runs, except where noted. The results of hyperparameter searches, which affected these values, appear in Appendix 4.

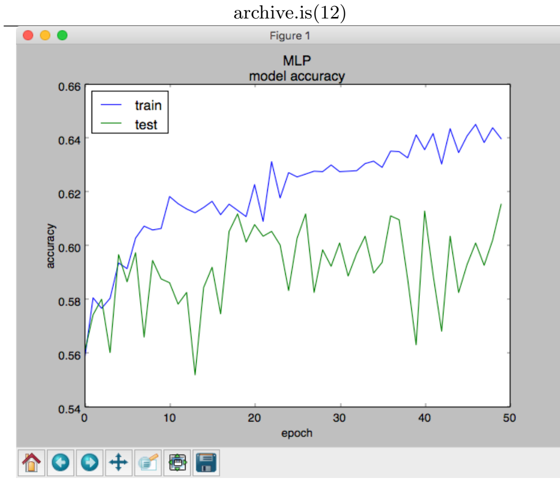
Character Convolutional Neural Network configurations used				
A	B	C	D	E
8 weight layers	8 weight layers	11 weight layers	8 weight layers	6 weight layers
input true and false url set x character dictionary				
Conv1-64	Conv1-24	Conv1-24	Conv1-256	Conv1-100
Conv1-64	Conv1-24	Conv1-24	Conv1-128	MaxPooling
Maxpooling	MaxPooling	Conv1-24	MaxPooling	Conv1-50
Conv1-128	Conv1-48	Conv1-24	Conv1-64	Dropout .5
Conv1-128	Conv1D-48	MaxPooling	Conv1-64	GlobalAveragePooling
GlobalAveragePooling	GlobalAveragePooling	GlobalAveragePooling	GlobalAveragePooling	Dense
Dropout .5	Dropout .5	Conv1-24	Dropout .5	
Dense	Dense	Dropout .5	Dense	
		Conv1-24		
		GlobalAveragePooling		
		Dense		

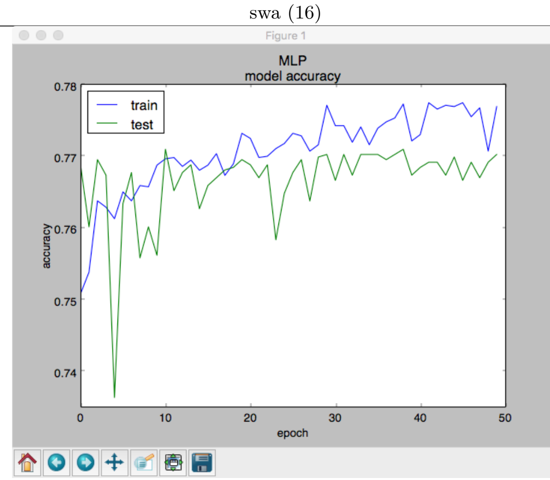
Table 9: A key to the various CNN architectures that were tested, Architecture A is identical to that used for spam classification

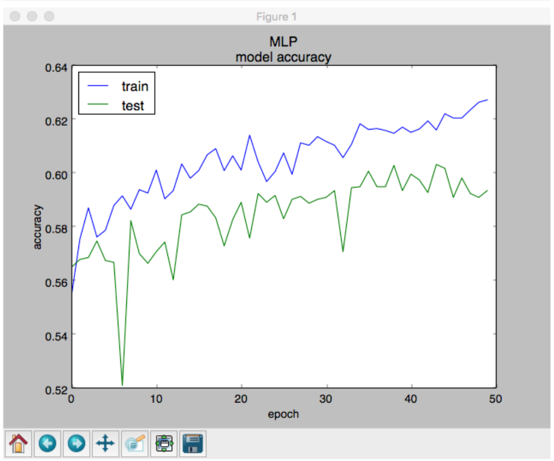
Char-MLP Neural Networks		
A	B	C
Dense (256 nodes)	Dense (256 nodes)	Dense (256 nodes)
Dense (256 nodes)	Dense (256 nodes)	Dense (128 nodes)
Dense (128 nodes)	Dense (128 nodes)	Dense (1-sigmoid)
Dense (128 nodes)	Dense (1-sigmoid)	
Dense (1-sigmoid)		

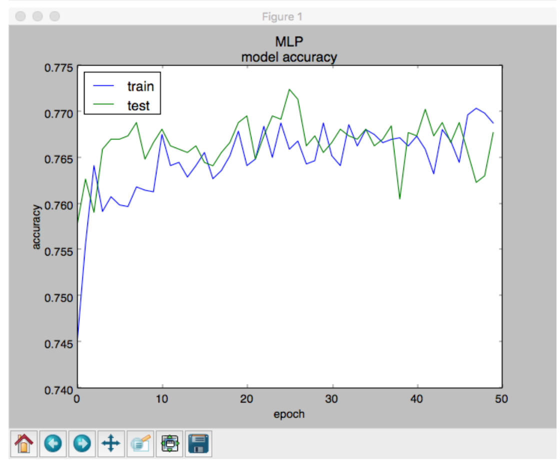
Table 10: Architecture of MLP networks that were evaluated

archive.is(12)	swa (16)









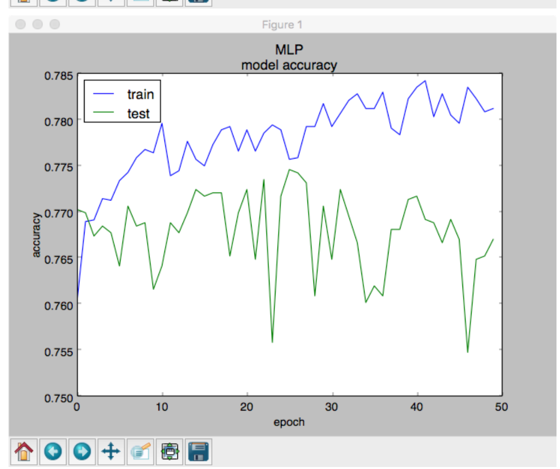


Table 11: Char-MLP training history for A,B,C with 6000 samples and a 70/30 split. When the lines remain close or converge, the learning and predictive performance of the model will be better.

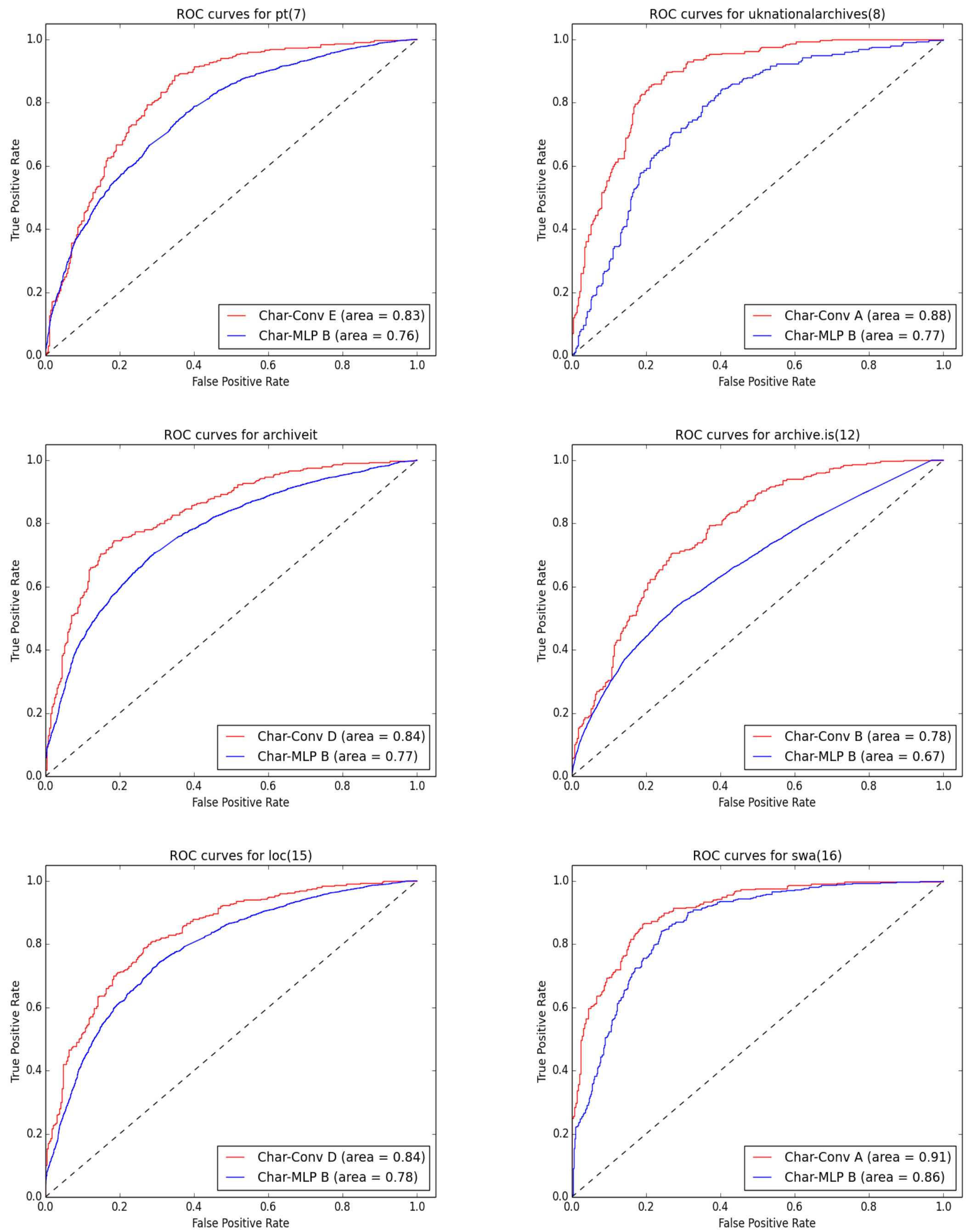
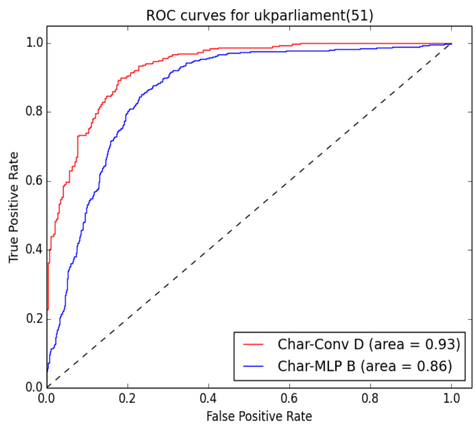
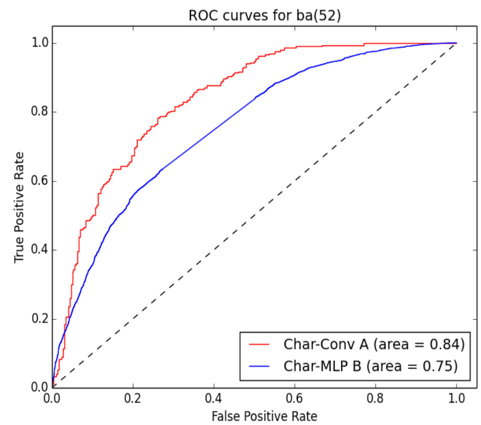


Table 12: Best results, MLP-B, and CNN curves: 20 epochs, batch size 32, 6000 samples





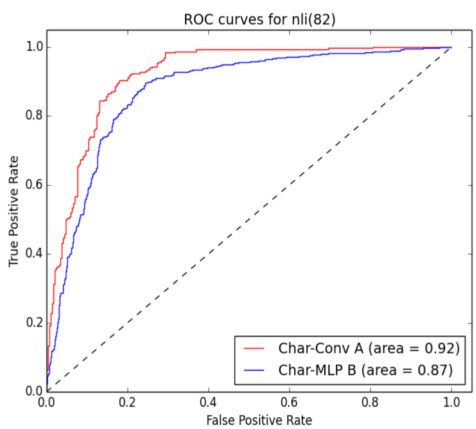


Table 13: Best results, MLP-B and CNN, continued

Network Architecture	GPU Run time	Mac Pro Run time
Char-CNN A	24.25	166.42
Char-CNN B	17.1	93.47
Char-CNN C	18.56	287.01
Char-CNN D	56.17	469.2
Char-CNN E	29.54	204.45
Stacked Char-CNN C	21.94	289.65
Char-MLP A	20.25	27.62
Char-MLP B	17.4	25.11
Char-MLP C	15.55	21.19

Table 14: Performance of various character neural network architectures, 2000 samples from archive id 7, 10 epochs, GPU and Mac Pro run times in seconds

	A	B	C	D	E	Stacked C
pt (7)	70.3	67.85	68.05	68.55	66.5	65.95
uknationalarchives (8)	73.65	72.7	71.45	71.6	69.4	70.75
archiveit (10)	65.15	67.8	66.85	66.8	70.4	65.25
archive.is (12)	62.3	60.55	61.75	64.15	60.7	60.75
loc (15)	68.85	69.3	67.0	71.25	66.6	68.5
swa (16)	74.2	76.55	77.9	74.35	76.5	72.95
proni (19)	77.9	79.35	79.2	78.4	80.0	79.6
ukparliament (51)	74.95	80.2	76.3	80.3	79.45	75.05
ba (52)	69.9	69.45	69.35	70.0	71.05	67.5
nli (82)	76.85	79.9	77.4	75.8	78.85	79.95
perma (86)	71.1	71.0	67.25	72.6	67.7	68.4

Table 15: Test results by archive id for six Char-CNN network architectures, 2000 samples, accuracy value is average of 10 executions

	A	B	C	D	E	Stacked C
pt (7)	67.7	67.93	70.95	68.73	70.73	68.08
uknationalarchives (8)	76.13	79.13	78.93	79.58	76.0	71.28
archiveit (10)	68.08	69.05	69.53	71.15	70.13	64.98
archive.is (12)	65.68	66.23	65.63	64.4	65.25	63.0
loc (15)	69.93	70.7	69.7	72.35	69.38	66.9
swa (16)	79.22	79.23	76.3	80.38	77.9	74.75
ukparliament (51)	81.6	82.23	80.4	79.33	80.25	79.68
ba (52)	70.43	71.38	70.15	71.68	70.65	70.4
nli (82)	78.95	79.13	78.93	79.58	79.7	78.7

Table 16: Test results by archive id for six Char-CNN network architectures, 4000 samples, accuracy is average of 10 executions

	A	B	C	D	E	Stacked C
pt (7)	70.62	70.57	69.4	72.03	68.57	68.05
uknationalarchives (8)	76.7	76.27	79.6	77.08	79.4	73.75
archiveit (10)	70.92	71.28	71.92	72.9	70.42	65.85
archive.is (12)	67.38	67.17	66.82	66.5	66.55	64.48
loc (15)	71.87	71.07	72.03	72.98	71.05	67.03
swa (16)	80.72	79.73	78.97	80.63	79.27	77.63
ukparliament (51)	81.33	81.67	81.87	82.25	80.3	80.65
ba (52)	72.82	72.3	70.95	72.93	72.22	70.65
nli (82)	80.32	80.33	79.6	81.37	79.4	77.58

Table 17: Test results by archive id for six Char-CNN network architectures, 6000 samples, accuracy is average of 10 executions

	A	B	C	D	E	Stacked C
pt (7)	71.64	70.71	69.28	70.32	71.06	68.01
uknationalarchives (8)	80.48	79.21	81.37	80.9	79.76	78.98
archiveit (10)	72.3	72.09	72.02	70.59	71.11	67.93
archive.is (12)	67.02	67.82	65.46	66.55	67.64	64.4
loc (15)	70.7	71.87	72.87	70.73	72.52	70.07
swa (16)	78.77	79.94	79.97	80.0	78.69	76.66
ukparliament (51)	82.11	81.36	82.22	81.51	79.47	80.35
ba (52)	72.53	73.23	73.27	73.94	71.34	71.79
nli (82)	80.48	79.21	81.37	80.9	79.76	78.98

Table 18: Test results by archive id for six Char-CNN network architectures, 10,000 samples, accuracy is average of 10 executions

	2000	4000	6000	10,000
pt (7)	64.95	68.7	67.98	68.92
uknationalarchives (8)	69.7	69.98	69.88	72.25
archiveit (10)	66.9	67.63	67.37	69.13
archive.is (12)	57.95	58.45	59.2	59.52
loc (15)	67.9	68.43	68.93	69.47
swa (16)	76.35	75.93	75.97	77.01
ukparliament (51)	79.9	78.85	78.87	79.38
ba (52)	62.6	65.63	65.55	66.35
nli (82)	80.5	78.83	78.92	78.79

Table 19: MLP test results by archive id 2000, 4000, 6000, and 10,000 samples

Evaluation

It should also be noted that the test data contained a large amount of variation in terms of number of URLs per archive ranging from a few hundred, to tens of thousands. Results for archives with smaller samples should be considered suspect, even though the true/false positive results were in line with those reported for other archives when the accuracy score was also similar. However, as table 17 illustrates, trends found in smaller samples tend to hold true in larger samples. These scatter plots represent projections of the URL-character matrices to a lower dimensional plot. Hits and misses for archive.is (id 12) appear to be distributed randomly, while there are readily discernible groupings of hits verses misses for ukparliament (id 51). This is no doubt why both Char-CNN and Char-MLP networks performed better learning over ukparliament data than archive.is data.

Models can be saved for re-use, including classification of previously unseen URLs. This functionality has been tested in a limited fashion. Starting with the 3000 sample Char-CNN network runs, the best model per cache id and network architecture, as determined by the Keras accuracy score using validation data, is saved for future testing and re-use. It would be helpful to have a discussion about how to test and evaluate saved models. On a Mac Pro, a model trained with Char-CNN B and 12,000 samples (6000 hits, 6000 misses) from archive.is was able to classify 50 randomly selected URLs in .997 seconds, 100 URLs in 1.769 seconds, and 200 URLs in 3.304 seconds. Based on this small test, it appears that this predictive model scales linearly.

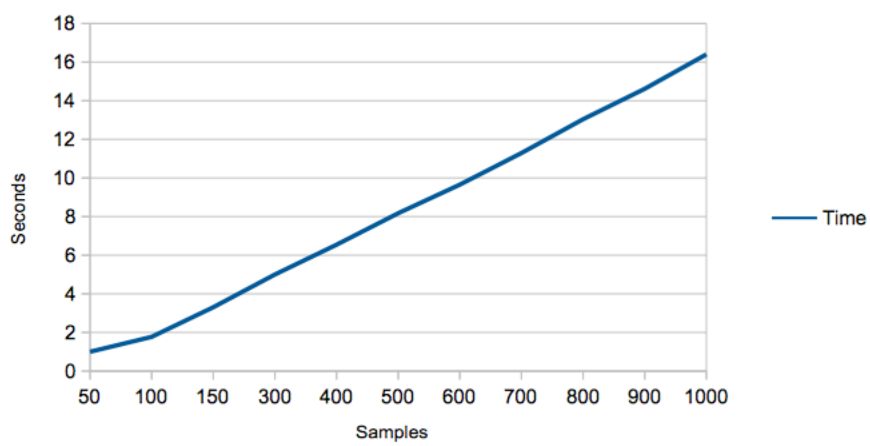


Figure 4: Char-CNN B model performance classifying 50 - 1000 random URLs

More data would be helpful, if these results merit the effort of extracting additional data. It is by no means certain that the architectures and hyperparameters that were tested were necessarily the best choices, even though every effort was made to evaluate options methodically. There may be other network architectures that may also yield good results, such as character recurrent network models such as Long Short Term Memory (LSTM) networks, which were not tested. Feature engineering might also improve results. URLs could be tokenized into words from which it might be useful to generate distributional co-occurrence vectors using Glove or Word2Vec. Again, whether or not additional work is appropriate depends on how these results compare with previous applications of machine learning to this problem.

References

- Leverington, David. A Basic Introduction to Feedforward Backpropagation Neural Networks http://www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html

- Jacobson, Lee. Introduction to Artificial Neural Networks http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-1/7

- Trask, Andrew. A Neural Network in 11 lines of Python https://iamtrask.github.io/2015/07/12/basic-python-network/

- Yadav, Vivek. How neural networks learn nonlinear functions and classify linearly non-separable data https://medium.com/@vivek.yadav/how-neural-networks-learn-nonlinear-functic

- Deshpande, Adit. A Beginners Guide to Understanding Convolutional Neural Networks https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

- Sharma, Avinash. Understanding Activation Functions in Neural Networks https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networl

- Comprehensive list of activation functions in neural networks with pros/cons https://stats.stackexchange.com/questions/115258/comprehensive-list-of-activation-functions-in-n

- Deshpande, Adit. The 9 Deep Learning Papers You Need to Know About https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-html

- Getting started with the Keras Sequential model https://keras.io/getting-started/sequential-model-guide/

- Brownlee, Jason. Rescaling Data for Maching Learning in Python with SciKit-Learn https://machinelearningmastery.com/rescaling-data-for-machine-learning-in-python-with-scik

- HTTP Framework for Time-Based Access to Resource States Memento https://tools.ietf.org/html/rfc7089

- Bornand, Nicolas J., Balakivera, L, Van de Sompel, H. Routing Memento Requests Using Binary Classifiers http://delivery.acm.org/10.1145/2920000/2910899/p63-bornand.pdf

- Watch Tiny Neural Nets Learn http://swanintelligence.com/watch-tiny-neural-nets-learn.html

- van de Maaten, Laurens. t-SNE https://lvdmaaten.github.io/tsne/

- An Illustrated Introduction to the t-SNE algorithm https://github.com/oreillymedia/t-SNE-tutorial

- GloVe: Global Vectors for Word Representation https://nlp.stanford.edu/projects/glove/

Appendix 1: Effects of more data on Char-MLP performance

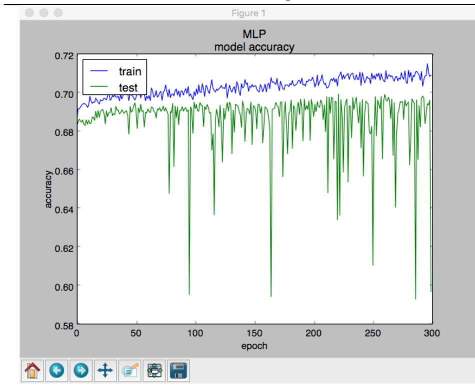
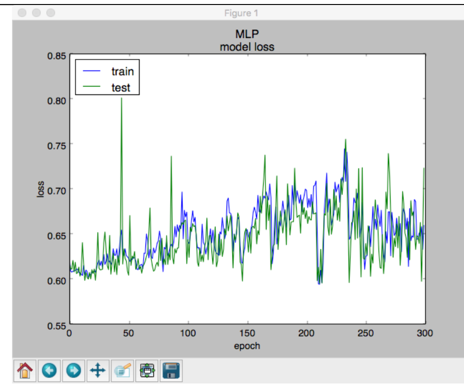
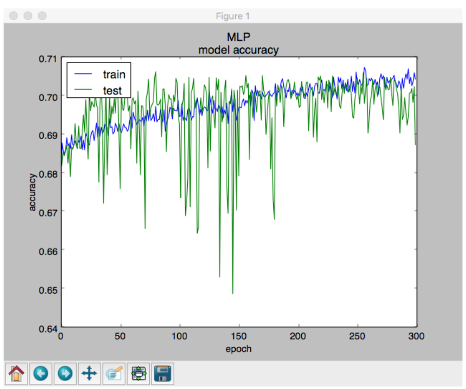
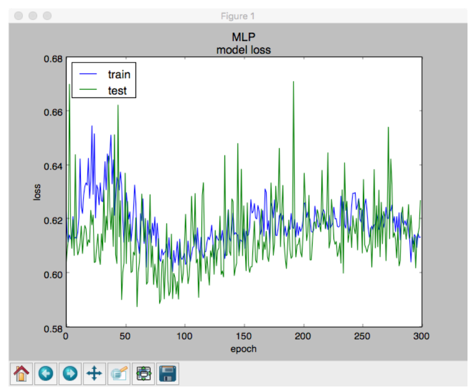
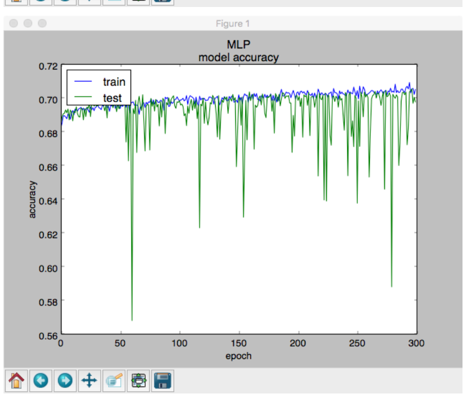
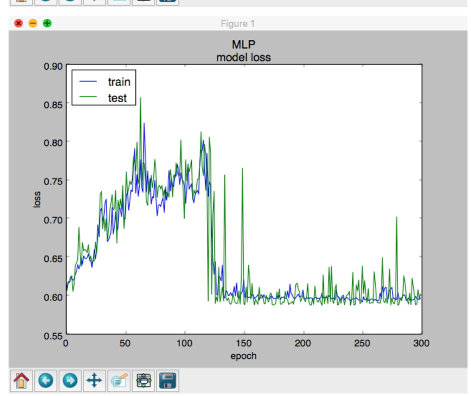
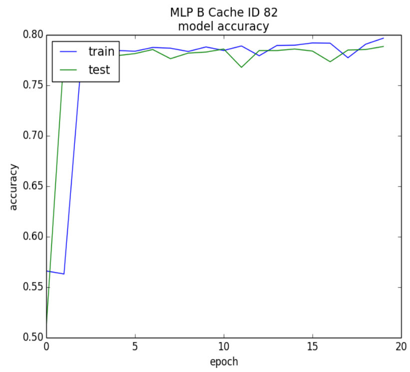
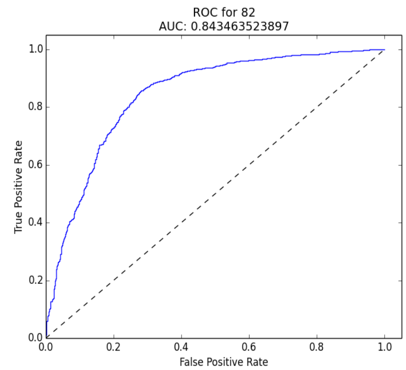
Training	Loss
	
	
	

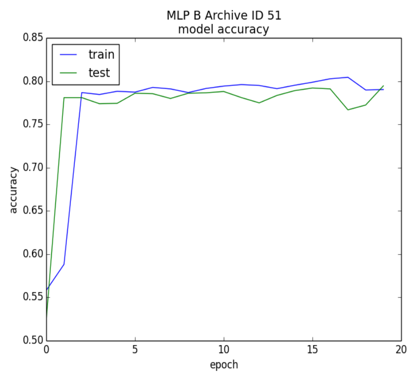
Table 20: Char-MLP: longer training (300 epochs) and progressively more samples (15000, 18000, 25000) training and loss.

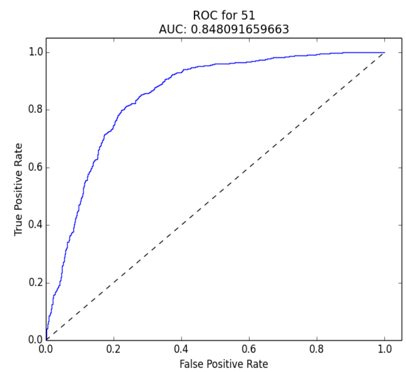
Appendix 2: Char-MLP runs that aim for optimal training results

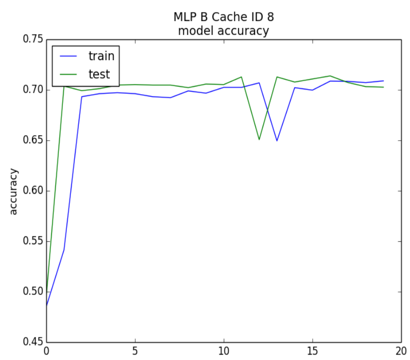
Training	ROC











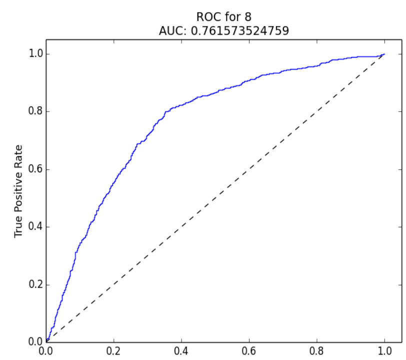


Table 21: Char-MLP B: smaller sample sets, fewer epochs, and larger batch size (50%) of sample size

Appendix 3: Char-CNN runs for all architectures

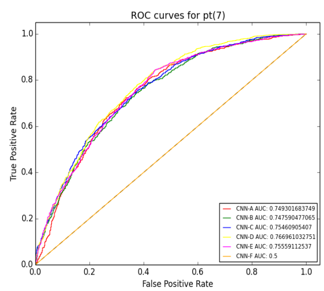
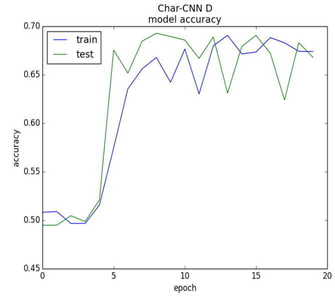
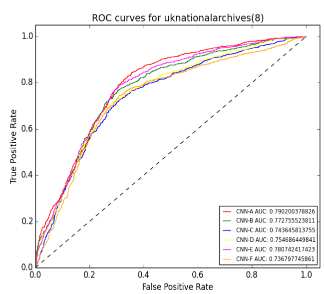
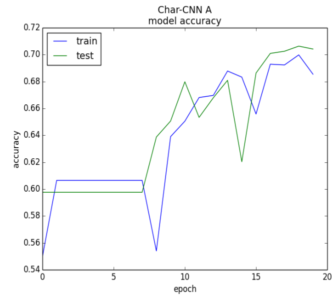
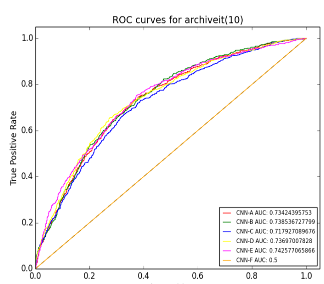
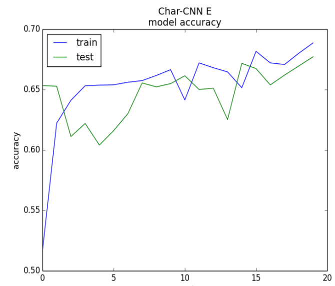
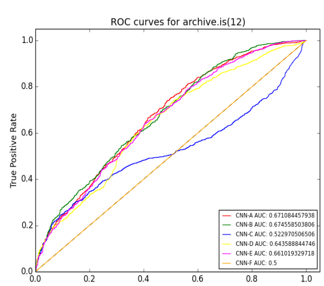
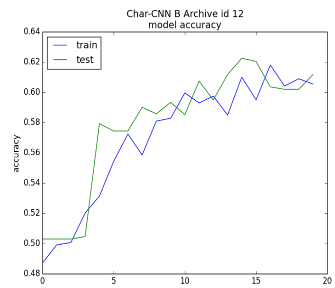
ROC	Training (best)
	
	
	
	

Table 22: Char-CNNs: 6000 samples, 20 epochs, batch size 1000

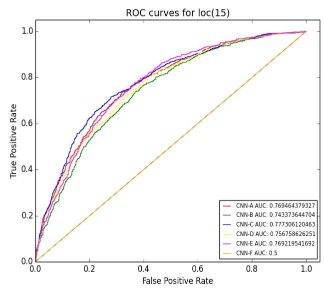
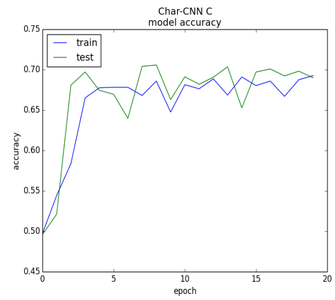
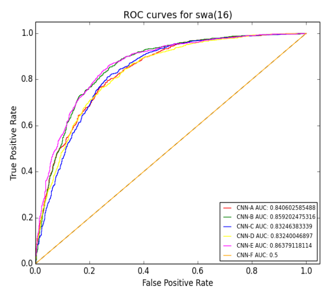
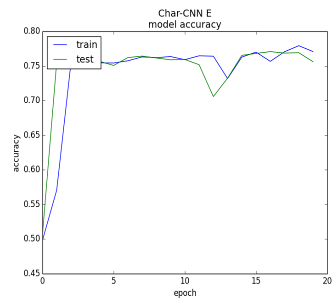
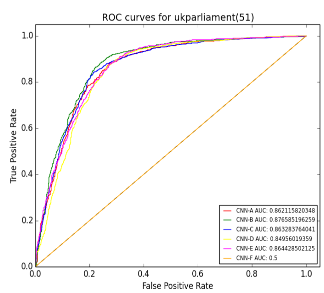
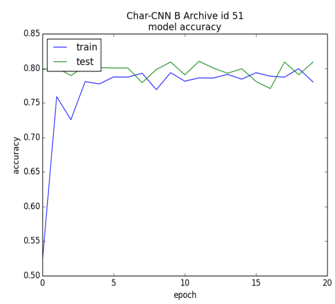
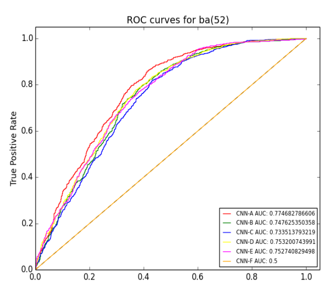
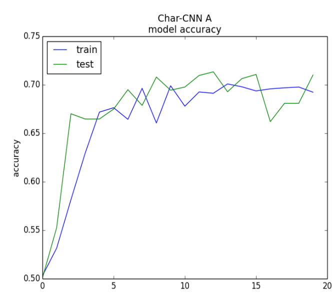
ROC	Training (best)
	
	
	
	

Table 23: Char-CNNs, continued

Appendix 4: Hyperparameter searching

Results						Network Parameters		
Archive ID	Accuracy	Correct	False Positive	Precision	Recall	Batch size	Epochs	Kernel size
7 (High)	70.7	141	22	.683	.777	20	10	4
7 (Low)	64.9	129	22	.643	.775	128	5	3
7 (Average)	68.74	137.02	23.38	.671	.762	N/A	N/A	N/A
8 (High)	75.1	150	18	.722	.833	48	5	3
8 (Low)	67.6	135	18	.665	.812	128	5	4
8 (Average)	72.46	144.6	18.53	.706	.809	N/A	N/A	N/A
10 (High)	70.45	140	29	0.715	.702	4	10	3
10 (Low)	63.9	127	25	.651	.741	64	5	4
10 (Average)	67.92	135.31	25.61	.676	.739	N/A	N/A	N/A
12 (High)	65.35	130	30	.647	.695	20	15	3
12 (Low)	55.6	111	42	.498	.58	128	5	4
12 (Average)	61.77	123.17	34.71	.62	.648	N/A	N/A	N/A
15 (High)	71.9	143	23	.706	.766	36	10	4
15 (Low)	62.6	125	29	.623	.708	116	10	3
15 (Average)	67.96	135.53	24.625	.67	.749	N/A	N/A	N/A
52 (High)	73.65	147	20	.719	.792	36	15	3
52 (Low)	66.25	132	26	.667	.731	128	5	4
52 (Average)	70.89	141.41	20.63	.688	.789	N/A	N/A	N/A

Table 24: Results for Char-CNN A executions using 2000 log entry samples per archive, with hyperparameters

Results						Network Parameters		
Archive ID	Accuracy	Correct	False Positive	Precision	Recall	Batch size	Epochs	Kernel size
7 (High)	71.88	431	85	.728	.714	20	10	3
7 (Low)	69.78	418	47	.657	.843	80	5	4
7 (Average)	70.45	422.45	70.27	.696	.764	N/A	N/A	N/A
8 (High)	78.01	468	43	.745	.855	32	5	3
8 (Low)	72.93	437	71	.65	.76	4	10	3
8 (Average)	76.7	459.81	51	.735	.828	N/A	N/A	N/A
10 (High)	73.36	440	66	.719	.777	32	5	4
10 (Low)	68.78	412	81	.706	.728	4	10	3
10 (Average)	71.69	429.6	73.7	.714	.752	N/A	N/A	N/A
12 (High)	68.06	408	81	.667	.728	20	10	3
12 (Low)	66.81	400	73	.65	.756	80	5	4
12 (Average)	66.66	399.54	85.63	.657	.712	N/A	N/A	N/A
15 (High)	72.63	435	78	.723	.737	20	10	3
15 (Low)	68.46	410	92	.727	.69	4	10	3
15 (Average)	71.08	425.9	64.7	.698	.782	N/A	N/A	N/A
16 (High)	80.85	485	40	.779	.864	32	5	4
16 (Low)	77.8	466	45	.756	.849	64	5	4
16 (Average)	78.79	472.3	46.4	.767	.843	N/A	N/A	N/A
51 (High)	83.06	498	35	.802	.88	32	5	3
51 (Low)	76.05	456	36	.745	.878	4	10	3
51 (Average)	81.07	486.09	35.9	.781	.878	N/A	N/A	N/A
52 (High)	73.16	439	63	.711	.789	32	5	3
52 (Low)	68.9	413	96	.734	.677	48	5	3
52 (Average)	71.34	427.72	59.27	.694	.801	N/A	N/A	N/A
82 (High)	80.96	485	51	.8	.827	20	10	3
82 (Low)	77.7	466	36	.747	.878	64	5	4
82 (Average)	79.76	478.09	41.72	.772	.859	N/A	N/A	N/A

Table 25: Results for Char-CNN A executions using 6000 log entry samples per archive, with hyperparameters

Results								Network Parameters		
Archive ID	Training Set	Validation set	Accuracy	Correct	False Positive	Precision	Recall	Batch size	Epochs	Kernel size
71 (High)	1000	100	83.2	83	4	0.794	.904	112	5	4
71 (Low)	1000	100	78.5	78	6	0.756	.896	64	5	4
71 (Average)	1000	100	81.537	81.094	4.735	.778	.896	N/A	N/A	N/A
19 (High)	2200	220	83.2	183	15	0.817	0.863	4	10	3
19 (Low)	2200	220	79.0	173	12	.751	.89	48	5	4
19 (Average)	2200	220	80.8	177.4	14	0.784	0.868	N/A	N/A	N/A
110 (High)	11000	1100	82.9	912	65	0.799	0.881	32	5	4
110 (Low)	11000	1100	80.6	886	59	0.766	0.892	96	5	3
110 (Average)	11000	1100	81.9	900.7	64.9	0.786	0.881	N/A	N/A	N/A
51 (High)	11000	1100	82.9	912	65	0.799	0.881	32	5	4
51 (Low)	11000	1100	77.8	856	48	0.736	0.911	4	10	3
51 (Average)	11000	1100	81.6	897.8	63.8	0.783	0.883	N/A	N/A	N/A
8 (High)		638	0.786	500	50	0.757	0.842	128	30	3
8 (Low)		638	0.742	474	63	0.77	0.692	128	30	3
8 (Average)		638	0.768	489.6	63	0.752	0.802	N/A	N/A	N/A
82 (High)		1286	0.829	1068	104	0.823	0.838	128	30	3
82 (Low)		1286	0.8	1029	186	0.865	0.71	128	30	3
82 (Average)		1286	0.813	1046.3	92.9	0.792	0.855	N/A	N/A	N/A
86 (High)		270	0.729	197	39	0.738	0.711	128	30	3
86 (Low)		270	0.511	139	132	0.8	0.029	128	30	3
86 (Average)		270	0.653	176.7	50.3	0.718	0.628	N/A	N/A	N/A
15 (High)	24000	2400	0.757	1817	231	0.733	0.807	32	5	3
15 (Low)	24000	2400	0.718	1725	160	0.668	0.866	32	5	3
15 (Average)	24000	2400	0.742	1782	268	0.733	0.776	N/A	N/A	N/A
16 (High)	9800	980	81.1	795	77	0.797	0.842	48	5	4
16 (Low)	9800	980	80.5	789	54	0.765	0.889	48	5	4
16 (Average)	9800	980	80.9	792.66	67.33	0.783	0.862	N/A	N/A	N/A
7 (High)	20000	2000	73.7	1475	219	0.723	0.780	32	5	4
7 (Low)	20000	2000	72.8	1456	236	0.724	0.763	32	5	3
7 (Average)	20000	2000	73.2	1465.5	227.5	0.723	0.772	N/A	N/A	N/A
12 (High)	24000	2400	0.720	1730	313	0.713	0.739	32	5	3
12 (Low)	24000	2400	0.673	1616	160	0.625	0.866	32	5	3
12 (Average)	24000	2400	0.698	1677	359	0.711	0.700	N/A	N/A	N/A
10 (High)	20000	2000	0.7358	1471	306	0.761	0.693	32	5	3
10 (Low)	20000	2000	0.7355	1470	248	0.734	0.7511	32	5	4
10 (Average)	20000	2000	.735	1470.5	277	0.747	0.722	N/A	N/A	N/A

Table 26: Results for Char-CNN A executions using as many samples as possible per archive, with hyperparameters

Results						Network Parameters		
Archive ID	Accuracy	Correct	False Positive	Precision	Recall	Batch size	Epochs	Kernel size
7 (High)	71.73	430	65	.695	.782	32	5	3
7 (Low)	67.96	407	54	.651	.819	128	5	3
7 (Average)	69.97	419.46	65.38	.681	.78	N/A	N/A	N/A
8 (High)	77.45	464	54	.752	.82	32	5	3
8 (Low)	74.26	445	48	.714	.838	128	5	3
8 (Average)	75.93	455.23	55.85	.739	.812	N/A	N/A	N/A
10 (High)	72.46	434	83	.733	.72	32	5	3
10 (Low)	66.88	401	118	.73	.605	128	5	3
10 (Average)	70.06	419.92	79.85	.707	.732	N/A	N/A	N/A
12 (High)	68.21	409	81	.669	.728	36	10	4
12 (Low)	62.43	374	94	.661	.685	96	5	4
12 (Average)	66.17	396.7	94.25	.671	.684	N/A	N/A	N/A
15 (High)	72.75	436	74	.721	.751	32	5	3
15 (Low)	68.81	412	32	.64	.892	96	5	4
15 (Average)	71.36	427.83	60.58	.692	.796	N/A	N/A	N/A
16 (High)	79.71	478	41	.766	.861	32	5	3
16 (Low)	77.3	463	37	.734	.873	112	5	4
16 (Average)	78.66	471.46	42.08	.756	.858	N/A	N/A	N/A
51 (High)	81.73	490	33	.779	.888	32	5	4
51 (Low)	79.21	475	38	.761	.871	96	5	3
51 (Average)	80.7	483.69	37.92	.775	.872	N/A	N/A	N/A
52 (High)	73.05	438	50	.694	.83	32	5	3
52 (Low)	70.55	423	45	.667	.848	128	5	3
52 (Average)	71.72	430.07	51.46	.683	.827	N/A	N/A	N/A
82 (High)	80.83	485	37	.772	.875	80	5	4
82 (Low)	77.2	463	31	.732	.893	112	5	4
82 (Average)	79.71	477.84	38.15	.763	.871	N/A	N/A	N/A

Table 27: Results for Char-CNN B executions using 6000 log entry samples per archive, with hyperparameters

Results						Network Parameters		
Archive ID	Accuracy	Correct	False Positive	Precision	Recall	Batch size	Epochs	Kernel size
7 (High)	70.36	422	77	.692	.743	48	5	4
7 (Low)	66.53	399	64	.656	.784	96	5	3
7 (Average)	68.63	411.38	67.85	.671	.772	N/A	N/A	N/A
8 (High)	76.58	459	42	.725	.858	32	5	3
8 (Low)	71.26	427	47	.691	.841	128	5	3
8 (Average)	74.75	448.07	55.77	.725	.812	N/A	N/A	N/A
10 (High)	71.16	427	83	.713	.721	80	5	4
10 (Low)	66.5	399	47	.632	.841	64	5	3
10 (Average)	68.75	412.16	71.75	.68	.759	N/A	N/A	N/A
12 (High)	65.75	394	78	.638	.739	32	5	3
12 (Low)	60.88	365	89	.614	.702	112	5	3
12 (Average)	63.2	378.76	97.54	.643	.672	N/A	N/A	N/A
15 (High)	72	432	75	.711	.747	48	5	4
15 (Low)	69.38	416	78	.692	.74	112	5	4
15 (Average)	70.63	423.53	67.31	.69	.774	N/A	N/A	N/A
16 (High)	78.53	471	51	.766	.829	32	5	3
16 (Low)	75.08	450	58	.743	.806	96	5	4
16 (Average)	76.7	459.75	48.58	.743	.836	N/A	N/A	N/A
51 (High)	82.18	493	34	.785	.886	32	5	3
51 (Low)	79.23	475	47	.773	.843	112	5	4
51 (Average)	80.9	485	35.69	.775	.879	N/A	N/A	N/A
52 (High)	72.16	433	56	.693	.812	32	5	4
52 (Low)	66.25	397	105	.685	.649	128	5	3
52 (Average)	71.06	425.92	60.23	.686	.797	N/A	N/A	N/A
82 (High)	79.76	478	39	.763	.867	32	5	3
82 (Low)	76.43	458	33	.725	.888	48	5	3
82 (Average)	77.75	465.92	41.62	.746	.859	N/A	N/A	N/A

Table 28: Results for Char-CNN C executions using 6000 log entry samples per archive, with hyperparameters